FM-Index

April 3, 2021

Contents

1 Overview

An FM-Index is a is a full text index that allows finding and counting the occurrences of a given string in a corpus. This is done via a Burrows-Wheeler and a Wavelet Tree along with a few auxiliary data structures.

2 Components

2.1 Burrows-Wheeler Transform

In short, performing a Burrows-Wheeler Transform[1] (BWT) on a string consists of sorting all possible rotations of that string and extracting the last character from each in order.

This is a reversible operation because the first column of the sorted set can be generated by rotating the last column (the BWT) into the first position and sorting. From there,

[1] Wikipedia: Burrows-Wheeler Transform

the columns can again be rotated and sorted resulting in the second column. This can be repeated until the entire string is reconstructed[2].

This process can be done in a more space efficient manner by noting that, if a stable sort is used, the reordering of rows is the same for every sort operation. Thus sorting the BWT gives a index-to-index mapping that, when walked, generates the original string.

2.1.1 Example:

Input: `hello-abbaca!`

Rotated:	Sorted:	
`hello-abbaca!`	`!hello-abbaca`	`a`
`!hello-abbaca`	`-abbaca!hello`	`o`
`a!hello-abbac`	`a!hello-abbac`	`c`
`ca!hello-abba`	`abbaca!hello-`	`-`
`aca!hello-abb`	`aca!hello-abb`	`b`
`baca!hello-ab`	`baca!hello-ab`	`b`
`bbaca!hello-a`	`bbaca!hello-a`	`a`
`abbaca!hello-`	`ca!hello-abba`	`a`
`-abbaca!hello`	`ello-abbaca!h`	`h`
`o-abbaca!hell`	`hello-abbaca!`	`!`
`lo-abbaca!hel`	`llo-abbaca!he`	`e`
`llo-abbaca!he`	`lo-abbaca!hel`	`l`
`ello-abbaca!h`	`o-abbaca!hell`	`l`

Output: `aoc-bbaah!ell`

2.2 Wavelet Tree

A Wavelet Tree[3] is a data structure that allows counting the occurrence of any given character in a given prefix of a string in $O(1)$ time and using $O(n)$ space. This is accomplished by way of a binary tree indexed on the bits of the character. Each layer of the tree hold one bit for each position in the input. Each node holds a bit vector indicating which child the corresponding positon becomes part of, along with periodic cached counts.

This tree, is stored in an array encoded in the same way as heap mapping is traditionally done. The left and right child of the node at position i are at positions $2i + 1$ and $2i + 2$.

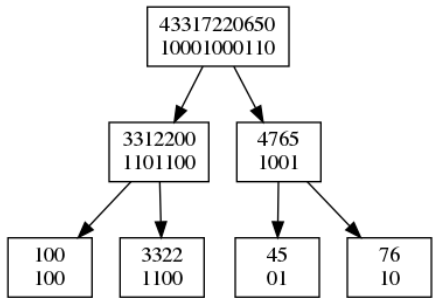
To generate a count, the tree is walked for the bits of the characters in question. At each node, the pop-count-before-position-i_n is found starting with a cached count and adding the popcount of the remaining locations. This sum then becomes the i_{n+1} at the next node.

Note that this works regardless of what character is at position i_0.

[2] Almost. A starting point is also needed to extract the correct rotation, but that's not needed in this case.

[3] Wikipedia: Wavelet Tree

2.2.1 Example:



3 FM-Index

An FM-index is composed of the BTW for the corpus being indexed and a wavelet tree of that BWT.

The FM-Index is inspected by considering the BWT as a representation of a hypothetical matrix containing all rotations of the corpus. To find a string (e.g. "abbaz"), it is walked backwards keeping track of the span of rows in the "matrix" that start with the suffix so far inspected. At the first step, every row starts with the empty string so the span is everything. At each subsequent step, the count of rows, above and inside the span (from the the prior step) which has as their last character the next character in the query is found. These counts, and the index of the first row starting with that character, give the span for the next step. This can be iterated backwards to test for any given string.

Once the target string is processed, the resulting span of locations can be converted to locations in the original string by walking from each location, using the BTW to find the preceding character, until a known location is found (e.g. via a table mapping the indexes of newlines or null character in the BTW to the corresponding location in the original corpus).

3.1 Optimizations

If the records to be indexed are un-ordered and unique (e.g. a set of strings) and none of them contain null character, then an optimization is possible that greatly simplifies the mapping back to the original records. If the corpus to be indexed is constructed by appending a null character to each record and concatenating them together, then when the BWT is constructed, the first n rows will result from the rotations that place the records, in sorted order, as starting in the second column. Given that, and a sorted list of the original records, then walking the BWT back one place past the next null character results in the index into that list of records.